C Defining SKOOL in K

SKOOL is a simple object-oriented language. It is defined as an extension of SILF with classes and objects. SKOOL is single-inheritance: except for the topmost builtin class `object`, each user-definable class in SKOOL extends precisely one other existing class; if an `extends` clause is not specified for a class, then it is assumed that that class extends `object`. Classes contain fields and methods. Objects can be created as instances of classes and messages can be send to objects under the form of method calls. Fields of an object are not visible directly from the outside of the object, but only by means of method calls. A `self` construct can be used as an object; it refers to the current object and it is particularly useful to achieve recursion. Method invocations can be preceeded by the keywork `super`, which redirects the call to the superclass; this is particularly useful when methods are overridden, to have access to methods in the superclass in case they were overridden by the current class. Object can be inspected for their type with an "instanceOf" language construct. SKOOL's methods have static scoping and dynamic dispatch. A SKOOL program consists of a set of classes. One class must be called `main` and must have a method called `main`. A program execution starts by creating an instance of the class `main` and then invoking the method `main` on that instance.

An untyped version of SKOOL is presented in Appendix C.1. Like in the case of untyped SILF, that means that variables and functions are are not declared any types. Nevertheless, the execution of programs gets stuck if certain incorrect operations are attempted, such as calling an inexistant method on an object, or adding an integer with a Boolean, etc. Appendix C.2 defines a dynamically typed variant of SKOOL, and Appendix C.3 discusses a static type checker for SKOOL taking into account subtype polymorphism. Here is a SKOOL program that can be executed by our semantics (both columns); the Maude output after executing this program is `2 2 3 11 3 3 11 111 111`:

```
class a {
  var i ; var j ;
  method a() {
    i = #1 ;
    j = i + #1 ;
    return j
  }
  method f() { return self . a() }
  method g() { return self . f() }
  method h() { return (i + j) }
}

class b extends a {
  var j ; var k ;
  method a() {
    return self . b()
  }
  method b() {
    call super a() ;
    j = #10 ;
    k = j + #1 ;
    return k
  }
  method g() { return super h() }
  method h() { return self . g() }
}

class c extends b {
  method a() { return super a() }
  method b() { return super b() }
  method c() {
    i = #100 ;
    j = i + #1 ;
    k = j + #1 ;
    return k
  }
  method g() { return (i + k * j) }
}

class main {
  method p(o) {
    write(o . f()) ;
    write(o . g()) ;
    write(o . h()) ;
    return #0
  }
  method main() {
    call self . p(new a()) ;
    call self . p(new b()) ;
    call self . p(new c()) ;
    return #0
  }
}
```

C.1 Untyped SKOOL

As mentioned above, we define SKOOL as an extension of SILF. We start by importing the K annotated syntax of untyped SILF *as is*, except for the syntax of function declaration and invocation constructs; we need to exclude the latter because we want functions to become methods and unrestricted function calls to become method invocation messages sent to individual objects. In some sense, a SKOOL program can be

regarded as a set of SILF programs, one per class, which can "communicate" with each other by means of method invocation messages sent to objects. We then add the following SKOOL-specific syntax:

K-Annotated Syntax of Untyped SKOOL

(include all syntax from SILF, except constructs for function declaration and invocation)

Exp	::=	...	
	|	new *Name*(*List*[*Exp*])	[*strict*(2)]
	|	self	
	|	*Exp*.*Name*(*List*[*Exp*])	[*strict*(1, 3)]
	|	super *Name*(*List*[*Exp*])	[*strict*(2)]
	|	*Exp* instanceOf *Name*	[*strict*(1)]
MethDecl	::=	method *Name*(*List*[*Name*]) *Stmt*	
	|	*MethDecl MethDecl*	$[md_1\ md_2 = md_1 \curvearrowright md_2]$
ClsDecl	::=	class *Name* extends *Name* {*VarDecl* ; *MethDecl*}	
	|	class *Name* extends *Name* {*MethDecl*}	[class *c* extends *c*′ {*md*} = class *c* extends *c*′ {· ; *md*}]
	|	class *Name* {*VarDecl* ; *MethDecl*}	[class *c* {*vd*; *md*} = class *c* extends object {*vd*; *md*}]
	|	class *Name* {*MethDecl*}	[class *c* {*md*} = class *c* extends object {*md*}]
	|	*ClsDecl ClsDecl*	$[cd_1\ cd_2 = cd_1 \curvearrowright cd_2]$

To define the K semantics of SKOOL, we also proceed by extending SILF's. However, in addition to the rules corresponding to the syntactic constructs for function declaration and invocation that have been replaced by method declaration and invocation, we also need to change a few other rules in SILF:

- The rules for variable (indexed or not) lookup and update need to change, because of the name resolution policy in object-orinted languages: if a name is not in the current environment (case in which the program would get stuck in SILF), then one should search for that name's location in the current class' layer in the current object; if not there, then one should move with the search up in the class hierarchy of that object, and so on until either the name is found or otherwise the topmost layer, object, is reached (in the latter case, the execution would get stuck).
- The rule for return also needs to change, because the structure of the stack needs to change. Indeed, the new stack will also push/pop the current class (in addition to the current environment and computation already considered in SILF).

Note that an object is a list of layers, each layer being an environment tagged with a class name. There is layer for each superclass of that object, as well as for its class; in other words, there is a layer for each class that the object can be regarded as an instance of. The semantics of object creation, as well as that of object and super method invocation, are all defined in terms of an auxilliary *invoke* computation item. When the first item in the computation, *invoke*(m, vl) starts searching for the method m with the current class. As it traverses the class hierarchy, the current class is also changed. This is very important, because we want static scoping in our language; therefore, once the method m is found and invoked, any variable should be lookup first in the current environment, and then in the class hierarchy of the current object starting with the class where the method was found! Note that that is indeed what our new definition for variable lookup does (using the auxilliary and easy to define operation *getLoc*).

The new $c(vl)$ construct first creates an object instance of class c and then calls the constructor method of c with arguments vl; constructor methods have the same name as their corresponding classes and are typically intended to initialize the values of the fields. The object creation process allocates to values in the store; it only creates an object layered environment, mapping each field in each layer to a fresh location. Note that the needed computation when returning from the construct invocation is stored in the stacked computation in anticipation: the return value of the construct is discarded (constructs are called for their side effects), then the current object is placed in the computation, and then the current object and computation are restored. As in SILF, methods are expected to return explicitly (using return statements). Objects and

super method invocations are easier to define than new. We prefer not to stack the current object because super does not change the current object, so it does not need to stack it.

K Configuration and Semantics of Untyped SKOOL

$$
\begin{array}{rcll}
\mathit{Val} & ::= & \ldots \mid \mathit{Object} & \\
\mathit{Object} & ::= & \mathit{List}[\mathit{Name} : \mathit{Env}] & \\
\mathit{Stack} & ::= & \mathit{List}[\mathit{Name} \times \mathit{Env} \times K] & \\
\mathit{ConfigItem} & ::= & k(K) \mid \mathit{stack}(\mathit{Stack}) \mid \mathit{env}(\mathit{Env}) \mid \mathit{in}(\mathit{List}[\mathit{Int}]) \mid \mathit{out}(\mathit{List}[\mathit{Int}]) \mid \mathit{store}(\mathit{Store}) \mid \mathit{nextLoc}(\mathit{Loc}) & \\
 & \mid & \mathit{obj}(\mathit{Object}) \mid \mathit{class}(\mathit{Name}) \mid \mathit{pgm}(K) & \\
\mathit{Config} & ::= & \mathit{List}[\mathit{Int}] \mid [\![K, \mathit{List}[\mathit{Int}]]\!] \mid [\![\mathit{Set}[\mathit{ConfigItem}]]\!] & \\
 & \mid & [\![K]\!] & [[\![k]\!] = [\![k, \cdot]\!]] \\
\mathit{KLabel} & ::= & \ldots \mid \mathit{create} \mid \mathit{addOEnvLayer} \mid \mathit{invoke} & \\
\mathit{KConstant} & ::= & \ldots \mid \mathit{restore}(\mathit{Object}) & [k(v \curvearrowright \underset{\cdot}{\underline{\mathit{restore}(o)\rangle}}\ \underset{o}{\mathit{obj}(\_)}]
\end{array}
$$

$[\![\mathit{cls}, \mathit{il}]\!] = [\![k(\mathsf{new\ main}())\ \mathit{stack}(\cdot)\ \mathit{env}(\cdot)\ \mathit{obj}(\cdot)\ \mathit{class}(\mathsf{main})\ \mathit{pgm}(\mathit{cls})\ \mathit{in}(\mathit{il})\ \mathit{out}(\cdot)\ \mathit{store}(\cdot)\ \mathit{nextLoc}(\mathit{loc}(0))]\!]$
$[\![\langle k(o)\ \mathit{out}(\mathit{il})\rangle]\!] = \mathit{il}$

$k(\underset{o}{\underline{\mathsf{self}\rangle}}\ \mathit{obj}(o)$
$\langle c : \_\rangle\ \mathsf{instanceOf}\ c = \mathsf{true}$
$o\ \mathsf{instanceOf}\ c = \mathsf{false}$, otherwise (i.e., when o has no layer labeled c)

$k(\underset{\mathit{create}(c) \curvearrowright \mathit{invoke}(c, \mathit{vl})}{\underline{\mathsf{new}\ c(\mathit{vl}) \curvearrowright k}})\ \mathit{env}(\underset{\cdot}{\underline{\rho}})\ \mathit{obj}(\underset{\cdot}{\underline{o}})\ \mathit{class}(\underset{c}{\underline{c'}})\ \mathit{stack}(\underset{(c', \rho, \mathit{discard} \curvearrowright \mathsf{self} \curvearrowright \mathit{restore}(o) \curvearrowright k)}{\underline{\cdot\rangle}})$
$k(\underset{\mathit{invoke}(m, \mathit{vl})}{\underline{(c : \eta, \omega).m(\mathit{vl}) \curvearrowright k}})\ \mathit{env}(\underset{\cdot}{\underline{\rho}})\ \mathit{obj}(\underset{(c : \eta, \omega)}{\underline{o}})\ \mathit{class}(\underset{c}{\underline{c'}})\ \mathit{stack}(\underset{(c', \rho, \mathit{restore}(o) \curvearrowright k)}{\underline{\cdot\rangle}})$
$k(\underset{\mathit{invoke}(m, \mathit{vl})}{\underline{\mathsf{super}\ m(\mathit{vl}) \curvearrowright k}})\ \mathit{env}(\underset{\cdot}{\underline{\rho}})\ \mathit{class}(\underset{c'}{\underline{c}})\ \mathit{stack}(\underset{(c, \rho, k)}{\underline{\cdot\rangle}})\ \mathit{pgm}\langle\mathsf{class}\ c\ \mathsf{extends}\ c'\ \{\_\ ;\ \_\}\rangle$

$k(\underset{\mathit{vd} \curvearrowright \mathit{addOEnvLayer} \curvearrowright \mathit{create}(c')}{\underline{\mathit{create}(c)\rangle}})\ \mathit{pgm}\langle\mathsf{class}\ c\ \mathsf{extends}\ c'\ \{\mathit{vd}; \mathit{md}\}\rangle$
$\mathit{create}(\mathsf{object}) = \cdot$
$k(\underset{\mathit{bind}\ \mathit{xl}\ \mathit{to}\ \mathit{vl} \curvearrowright s}{\underline{\mathit{invoke}(m, \mathit{vl})}})\ \mathit{class}(c)\ \mathit{pgm}\langle\mathsf{class}\ c\ \mathsf{extends}\ \_\ \{\_\ ;\ \langle\mathsf{method}\ m(\mathit{xl})\ s\rangle\}\rangle$
$k(\mathit{invoke}(m, \mathit{vl})\rangle\ \mathit{class}(\underset{c'}{\underline{c}})\ \mathit{pgm}\langle\mathsf{class}\ c\ \mathsf{extends}\ c'\ \{\_\ ;\ \_\}\rangle$, otherwise (i.e., when c has no method m)
$k(\underset{v \curvearrowright k}{\underline{\mathsf{return}(v) \curvearrowright \_}})\ \mathit{class}(\underset{c}{\underline{\_}})\ \mathit{env}(\underset{\rho}{\underline{\_}})\ \mathit{stack}(\underset{\cdot}{\underline{(c, \rho, k)\rangle}})$

$k(\underset{\sigma[\mathit{getLoc}(x, (\_ : \rho, c : \eta, \omega'))]}{\underline{x\rangle}})\ \mathit{env}(\rho)\ \mathit{class}(c)\ \mathit{obj}(\omega, c : \eta, \omega')\ \mathit{store}(\sigma)$
$k(\underset{\sigma[\mathit{getLoc}(x, (\_ : \rho, c : \eta, \omega')) +_{\mathit{Loc}} n]}{\underline{x[n]\rangle}})\ \mathit{env}(\rho)\ \mathit{class}(c)\ \mathit{obj}(\omega, c : \eta, \omega')\ \mathit{store}(\sigma)$
$k(\underset{\cdot}{\underline{x = v\rangle}})\ \mathit{env}(\rho)\ \mathit{class}(c)\ \mathit{obj}(\omega, c : \eta, \omega')\ \mathit{store}(\underset{\sigma[\mathit{getLoc}(x, (\mathit{env} : \rho, c : \eta, \omega')) \leftarrow v]}{\underline{\sigma}})$
$k(\underset{\cdot}{\underline{x[n] = v\rangle}})\ \mathit{env}(\rho)\ \mathit{class}(c)\ \mathit{obj}(\omega, c : \eta, \omega')\ \mathit{store}(\underset{\sigma[\mathit{getLoc}(x, (\mathit{env} : \rho, c : \eta, \omega')) +_{\mathit{Loc}} n \leftarrow v]}{\underline{\sigma}})$

$\mathit{getLoc}(x, (\_ : \eta, \omega)) = \eta[x]$ when $\eta[x]$ is defined
$\mathit{getLoc}(x, (\_ : \eta, \omega)) = \mathit{getLoc}(x, \omega)$ when $\eta[x]$ is not defined